TerraNNI: Natural Neighbor Interpolation on a 3D Grid Using a GPU*

Alex Beutel
School of Computer Science
Carnegie Mellon University
abeutel@cs.cmu.edu

Thomas Mølhave, Pankaj K. Agarwal
Department of Computer Science
Duke University
{thomasm,pankaj}@cs.duke.edu

Arnold P. Boedihardjo, James A. Shine
Topographic Engineering Center
U.S. Army Corps of Engineers
{Arnold.p.boedihardjo,James.a.shine}@usace.army.mil

ABSTRACT

With modern focus on LiDAR technology the amount of topographic data, in the form of massive point clouds, has increased dramatically. Furthermore, due to the popularity of LiDAR, repeated surveys of the same areas are becoming more common. This trend will only increase as topographic changes prompt surveys over already scanned terrain, in which case we obtain large spatio-temporal data sets.

In dynamic terrains, such as coastal regions, such spatio-temporal data can offer interesting insight into how the terrain changes over time. An initial step in the analysis of such data is to create a digital elevation model representing the terrain over time. In the case of spatio-temporal data sets those models often represent elevation on a 3D volumetric grid. This involves interpolating the elevation of LiDAR points on these grid points. In this paper we show how to efficiently perform natural neighbor interpolation over a 3D volumetric grid. Using a graphics processing unit (GPU), we describe different algorithms to attain speed and GPU-memory trade-offs. Our algorithm extends to higher dimensions. Our experimental results demonstrate that the algorithm is efficient and scalable.

Categories and Subject Descriptors: D.2 [Software]: Software Engineering; F.2.2 [Nonnumerical Algorithms and Problems]: Geometrical problems and computations; H.2.8 [Database Management]: Database Applications—*Data Mining, Image Databases, Spatial Databases and GIS*

General Terms: Performance, Algorithms

Keywords: LIDAR, Massive data, GIS, Natural Neighbor Interpolation

*This work is supported by NSF under grants CNS-05-40347, IIS-07-13498, CCF-09-40671, and CCF-1012254, by ARO grants W911NF-07- 1-0376 and W911NF-08-1-0452, by U.S. Army ERDC-TEC grant W9132V-11-C-0003, by NIH grant 1P50-GM-08183-01, and by a grant from the U.S.-Israel Binational Science Foundation..

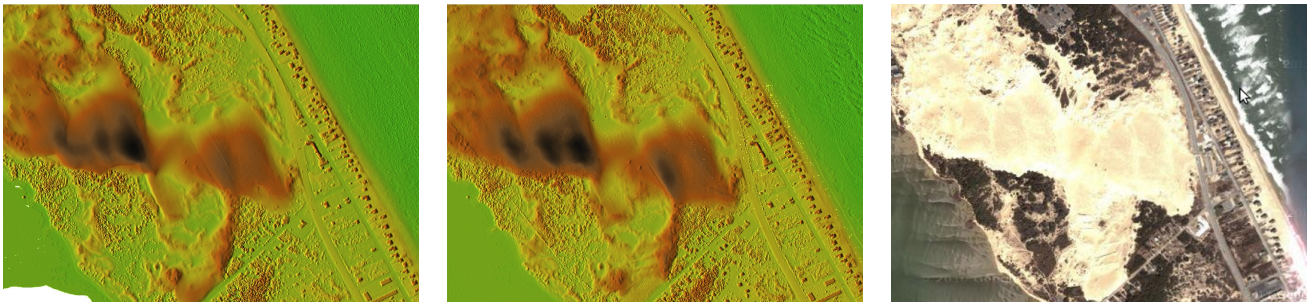
ACM SIGSPATIAL GIS '11 November 1-4, 2011. Chicago, IL, USA

1 Introduction

Modern sensing and mapping technologies such as airborne LiDAR technology can rapidly map the earth's surface at 15-20cm horizontal resolution. Since LiDAR provides a relatively simple and cheap way of surveying large terrains, some areas are (or will be) surveyed multiple times, yielding large spatio-temporal data sets. For example, the coastal region of North Carolina has been mapped every 1-2 years since the mid 1990's[1]. Figure 1 shows a spatio-temporal model of a portion of this region constructed from the data set. Such data sets provide tremendous opportunities for a wide range of commercial, scientific, and military applications. For instance, unmanned aerial vehicles can survey an area over a certain time period and the resulting spatio-temporal topographic data can be used to detect the location of new buildings, machinery, or vegetation. On a larger scale, they can be used to offer interesting insights into understanding terrain dynamics. For example, there has been much interest on studying the movement of sand dunes and erosion at the North Carolina coast [18]. It is essential for many applications to exploit the high-resolution data sets since small features may have a large impact on the output. For instance, it is vital that dikes and other features are present in data sets used for hydrological modeling, but these features are often relatively small and unlikely to disappear if the resolution is low [5, 19].

Capitalizing on the opportunities presented by high-resolution data sets and transforming massive amount of spatio-temporal topographic data into useful information requires that several algorithmic challenges be addressed. To begin with, the scattered point set S generated by LiDAR cannot be used directly by many GIS algorithms. They instead operate on a digital elevation model (DEM). Because of its simplicity and efficiency, the most widely used DEM is a 2D uniform grid in which an elevation value is stored at each cell. For spatio-temporal data, this translates into a 3D grid with time representing the third dimension. One has to extend the elevation measured by LiDAR at the points of S via interpolation to a uniform grid $\mathbb{Q} \subset \mathbb{R}^3$ of the desired resolution; the interpolation is performed in both time and space. We note that the need to interpolate a spatio-temporal data set on a uniform grid arises in a variety of applications. For example, one may want to interpolate data generated by a sensor network deployed over a wide area [9].

There is extensive work on statistical modeling of spatio-temporal data in many disciplines, including geostatistics, environmental sciences, GIS, atmospheric science, and biomedical engineering. It



(a) Interpolated DEM in 2002 (b) Interpolated DEM in 2005 (c) Aerial photo of the dune

Figure 1. Jockey's Ridge State Park sand dune in Nags Head, NC, USA.

is beyond the scope of this paper to discuss these methods here. We refer to [12, 14, 24] for reviews of many such results. In the context of GIS, interpolation methods based on kriging, inverse distance weighting, shape functions, random Markov fields, and splines have been proposed; see [17, 22, 13, 15] and references therein. Although sophisticated methods such as kriging and RST (spline based method [17]) produce high quality output, especially when data is sparse, they are computationally expensive and not scalable. On the other hand, simple methods such as constructing triangulation on input points in $\mathbb{R}^3$ [13] and linearly interpolating the elevation on grid points, though efficient, generate artifacts in the output and are not suitable for many GIS applications.

In this paper we use the well-known *natural neighbor interpolation* (NNI) strategy [23]. Given a finite set S of points in $\mathbb{R}^d$, a height function $h : S \to \mathbb{R}$ can be extended to the entire $\mathbb{R}^d$ using natural neighbor interpolation. In particular, for a point $q \in \mathbb{R}^d$,

$$h(q) = \sum_{p \in S} w_p(q) h(p), \tag{1}$$

where $w_p(q) \in [0, 1]$ is the fractional volume of $\mathrm{Vor}_{S \cup \{q\}}(q)$ that belongs to $\mathrm{Vor}_S(p)$ (see Figure 2 for a 2D example), i.e.,

$$w_p(q) = \frac{\mathrm{Vol}(\mathrm{Vor}_S(p) \cap \mathrm{Vor}_{S \cup \{q\}}(q))}{\mathrm{Vol}(\mathrm{Vor}_{S \cup \{q\}}(q))}, \tag{2}$$

where $\mathrm{Vor}_A(z)$ denotes the Voronoi cell of the point $z \in A$ in the Voronoi diagram of A. See Section 3 for a formal definition of the Voronoi diagram. NNI produces a smoother surface than linear interpolation and readily extends to spatio-temporal data. However, computing NNI on grids in 3D is challenging and expensive. First, the size of the Voronoi diagram in 3D can be quadratic in the worst case even for the Euclidean metric [6]. As mentioned below, for spatio-temporal data, we may have to compute the Voronoi diagram under more general metrics. Even if the size of the Voronoi diagram is near linear for realistic data sets [3], computing it is still expensive, especially on large data sets that do not fit in main memory. Since S can not be assumed to be in general position, robust implementations must take great care to handle cases such as point duplicates and groups of multiple points on the same plane/sphere; the existing implementations of Voronoi diagram construction such as Qhull are slow on degenerate inputs. Furthermore, performing NNI involves computing the intersection of two polyhedra and the volume of this intersection. Hemsley [10] has implemented NNI in 3D under the Euclidean metric, but the implementation is not scalable; see Section 5 for more details. We are unaware of any robust implementation of NNI for points in $\mathbb{R}^3$ that can handle large data sets. Because of these challenges, we propose to discretize the Voronoi diagram and compute NNI approximately; the approximation error can be controlled by choosing discretization parameters appropriately.

To attain significant speed-up in the NNI computation, we exploit the graphics processing units (GPUs) available on modern PCs. Although originally designed for quickly rendering 3D geometric scenes on an image plane (screen) and extensively used in video games, they can be regarded as massively parallel vector processors suitable for general purpose computing. Known as general purpose GPUs (GPGPUs), their tremendous computational power and memory bandwidth make them attractive for applications far beyond the original goal of rendering complex 3D scenes. As GPUs have become more flexible and programmable (e.g. NVIDIA's CUDA [20] library) they have been used for a wide range of applications, e.g., geometric computing, robotic collision detection, database systems, fluid dynamics, and solving sparse linear systems; see [21] for a recent survey. In the context of grid DEM construction, Fan *et al.* [8] and Beutel *et al.* [5] have described a GPU based algorithm for (NNI) in 2D.

Our contributions. Let $S \subseteq \mathbb{R}^3$ be a set of LiDAR points on which elevation is measured, and let $\mathbb{Q}$ be a 3D grid of points at which we want to compute the elevation. We present a simple yet very fast GPU based algorithm, called TerraNNI for performing a variant of NNI, which is more suitable for our application, on the points of $\mathbb{Q}$. Our algorithm is a generalization of the algorithm described in [5], but several new ideas are needed in 3D. LiDAR scanners provide dense point cloud of elevation data at most locations, but there are gaps. In space, these gaps usually appear at large bodies of water or human-made objects that have been removed from the point cloud in a preprocessing step. In time, the gaps appear when surveys fail to cover exactly the same region as previous surveys. When these gaps are large (i.e., there is a large region in space *and* time with no data) we wish to label the corresponding *gap* cells in the volumetric grid with *nodata* instead of interpolating elevation based on points that are very far away. We extend the notion of the *region of influence* of an input point as introduced in [5], to include time. We compute the elevation at a grid point using only those input points that lie within its region of influence. Our algorithm allows the "radius" of region of influence to be arbitrarily large. If it is set sufficiently large then the algorithm computes the standard NNI.

Since we are working with spatio-temporal data, Euclidean distance may not necessarily be the appropriate function to measure the distance between two points. We therefore assume that we have a metric $d(\cdot, \cdot)$ and compute Voronoi diagrams under this metric. While computing Voronoi diagrams on the CPU for general metrics is quite hard, we show that it is relatively straightforward on a GPU. We also show how the computation of Voronoi diagrams can be optimized for "weighted" Euclidean distance.

The main difficulty in performing NNI on the points of the 3D

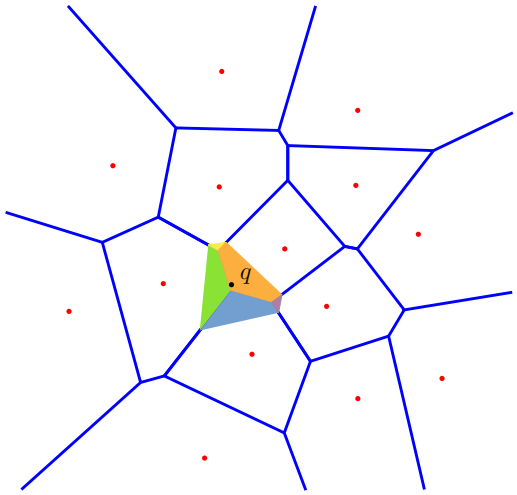


Figure 2. Natural neighbor interpolation for a point set S and query point q in $\mathbb{R}^2$. The shaded cell is $\mathrm{Vor}_{S\cup\{q\}}(q)$, and each color denotes the area stolen from each cell of $\mathrm{Vor}(S)$.

grid $\mathbb{Q}$ is that GPUs support only two-dimensional buffers, so unlike in [5], we cannot perform NNI on all grid points in one pass. Instead, we handle one 2D slice of $\mathbb{Q}$ at a time. This involves processing each input point several times. We present three algorithms, which provide trade offs between the number of times each point is processed and the space (on the GPU) used by the algorithm. The first algorithm stores only one copy of the GPU frame buffer but processes each point $O(r^2)$ times, where r is the "radius" of influence along the time axis, see Section 3 for a precise definition (here we assume that the temporal resolution of $\mathbb{Q}$ is 1). The second algorithm reduces the number of passes over the input points to $O(r)$ by storing r copies of the GPU frame buffer. Finally, if the input data is time-series data, i.e., $S = \Sigma^1, \ldots, \Sigma^T$ for $|T| \ll |S|$, where all points in Σ^i have the same time coordinate, and if the distance $d(\cdot,\cdot)$ is a weighted Euclidean metric (see Section 3 for the definition), we describe an algorithm that makes only one pass through the input points, under the assumption that some primitive operations on the GPU buffers can be performed; see Section 4 for details. The algorithm extends to certain other metrics as well, but we omit this extension from this version of the paper. We refer to Table 1 in Section 4 for an overview of the relative performance of our algorithms.

Our experimental results demonstrate that TerraNNI is both scalable and efficient. It interpolates over 450 million input points spanning 2 years (8.8GB on disk) in 26 minutes. We also test the memory trade-offs on a smaller data set consisting of about 20 million data points spanning 11 years. The simplest algorithm processing each point $O(r^2)$ times is 2-2.2 times slower than the algorithm that only processes each point $O(r)$ times at a cost of using r frame buffers. The latter took six and a half minutes to interpolate at 48 million points. We also tested the Interpolate3d [10] package which is an implementation of NNI on the CPU. Interpolate3d was unable to handle the full 20 million point data set with the 8GB of memory available in the machine, but was able to interpolate a 10 million point subset in about an hour and a half. We also compared TerraNNI against Qhull [4] and CGAL [2] which both contain state of the art Delaunay triangulation implementations, which are equivalent to the first step of performing NNI. It took CGAL eleven minutes to create the Delaunay triangulation of the 20 million input points, but it crashed when attempting to triangulate the larger 450 million point data set. Compare the eleven minutes for the smaller data set with the six and a half minute used by TerraNNI to compute the (discretized) Voronoi diagram *and* perform the interpolation at 48 million points. Out of those six and a half minutes, only 50 seconds where spent on loading the data and generating the Voronoi diagram. We refer to Section 5 for details.

Finally, we apply the grid DEM constructed by TerraNNI to analyze the terrain dynamics. In particular, we interpolate a multitemporal LiDAR of Nags Head, NC and study the movements of sand dunes in the coastal region.

Overview of the paper. This paper is organized as follows. Section 2 gives a short introduction to the fundamentals of the GPU model of computation, and Section 3 presents an algorithm for computing Voronoi diagrams in 3D. Section 4 presents the different algorithms for interpolating on volumetric 3D grids, and Section 5 compares the results from these different algorithms. Finally, we present the application on the NC coast data in Section 6 and present a short conclusion in Section 7.

2 GPU Model of Computation

In this section we give a brief overview of the parts of the model of computation offered by GPUs that are relevant for our paper.

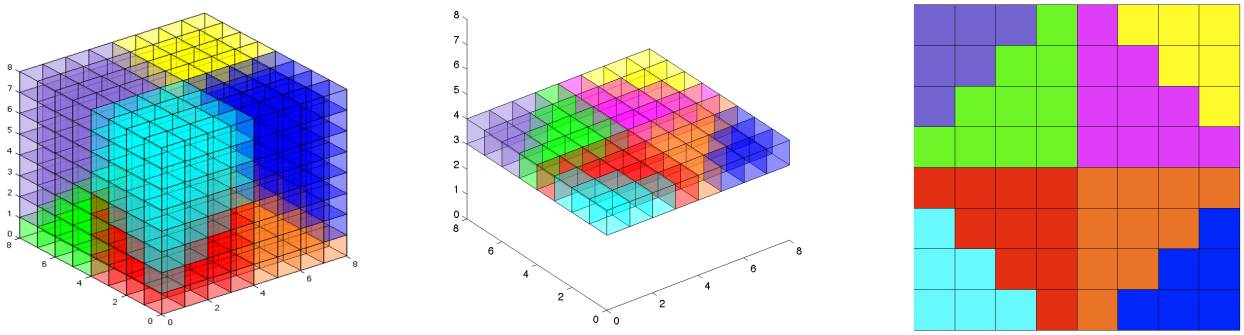
The graphics pipeline. The graphics pipeline is responsible for drawing 3D scenes, composed of many objects, onto a 2D image plane of pixels as seen from a viewpoint o. Because of their simplicity and flexibility, these objects are almost always triangles. For each pixel $\pi = (x, y)$ where x, y is a global coordinate, the GPU finds all objects $\Omega = \{\omega_1, \omega_2, \ldots \omega_n\}$ which ray $\vec{o\pi}$ intersects. To store the information about the scene, the GPU keeps two dimensional arrays of pixels called buffers:

- The *depth buffer* $\mathbb{D}$ stores the distance to the nearest object from o for each pixel π. Given that p_j is the point of intersection for ray $\vec{o\pi}$ and object ω_j, the GPU calculates $\mathbb{D}[\pi] = \min_{1\le j\le n} \|op_j\|$.
- The *color buffer* $\mathbb{C}$ stores the color of the scene as viewed from o. If for each object $\omega_j \in \Omega$ we have a color χ_j, we define a blending function that computes the color at each pixel: $\mathbb{C}[\pi] = \sum_{1\le j\le n} \alpha_j \chi_j$, where $\alpha_j \in [0, 1]$ is the blending parameter of ω_j. Typically, α_j is based on depth buffer so that $\mathbb{C}$ stores the color of the foremost object.

During graphical computations, the color and depth buffers reside in memory on the graphics card. Objects can be drawn onto these buffers with specific APIs such as OpenGL or Microsoft's DirectX. In our computations, there are cases where we will want to save and reuse buffers. Through the use of OpenGL-specific APIs, we can switch the buffer being used for drawing between multiple buffers stored in GPU memory. However, in some cases we will want to use the values in these buffers for computation on the CPU. For this, we have to read the buffer back to the computer's main memory. Unfortunately, since this involves transferring large amounts of data over the relatively slow bus systems, read backs are very slow. For using the GPUs parallel processing capabilities, popular graphics card manufacturer NVIDIA has created the CUDA parallel computing architecture [20], which facilitates general purpose parallel programming on the GPU.

3 Pixelized Voronoi Diagram

Let $S = \{p_1, \ldots, p_n\}$ be a set of points in $\mathbb{R}^3$. We view $\mathbb{R}^3$ to be xyt-space — x,y being spatial dimension and t being the time axis;



(a) (b) (c)

Figure 3. Pixelized Voronoi Diagram where $d(\cdot,\cdot)$ is the Euclidean metric. (a) $\mathrm{PVor}(S)$ for a set of eight points; (b,c) $\mathrm{PVor}_S(3)$, shown as a slice of $\mathbb{B}$ from (a).

each point $p = (x_p, y_p, t_p)$ is located in a 2-dimensional region, with (x_p, y_p) as its spatial coordinates, and has a time value t_p associated with it. We thus view S as spatio-temporal data. For convenience, we set $p_i = (x_i, y_i, t_i)$. For a point $p \in \mathbb{R}^3$, we use $p^* = (x_p, y_p)$ to denote its projection on the xy-plane. We also assume that we have a metric $d(\cdot,\cdot)$. Since we are dealing with spatio-temporal data, the Euclidean metric may not necessarily be the appropriate function to measure the distance between two points. For example, we may want to scale the t-axis differently from the x, y-axis and define

$$d(p,q) = \left((x_p - x_q)^2 + (y_p - y_q)^2 + \omega^2(t_p - t_q)^2\right)^{1/2} \qquad (3)$$

for some parameter $\omega \in \mathbb{R}$. We refer to this function as the weighted Euclidian metric.

Pixelized Voronoi diagram. For a point $p_i \in S$, we define its Voronoi cell $\mathrm{Vor}_S(p_i)$ to be

$$\mathrm{Vor}_S(p_i) = \{z \in \mathbb{R}^3 \mid d(z,p_i) \le d(z,p_j) \quad 1 \le j \le n\}.$$

$\mathrm{Vor}(S)$ is the subdivision of $\mathbb{R}^3$ induced by the Voronoi cells of points in S.

Let $\mathbb{B}$ be an axis-aligned box in $\mathbb{R}^3$ whose projection on the xy-plane is a square. We are interested in computing a discretized version of $\mathrm{Vor}(S)$ inside $\mathbb{B}$, defined below. We discretize $\mathbb{B}$ into a $N_s \times N_s \times N_t$ uniform 3D grid of voxels; N_s is the spatial resolution of the grid and N_t is its temporal resolution. Each voxel is a (tiny) box and its volume is $\mu = \mathrm{Vol}(\mathbb{B})/(N_s^2 N_t)$. We use $\hat{v}$ to denote the center-point of voxel v and index the voxels as (i,j,t) for $0 \le i, j < N_s$ and $0 \le t < N_t$. With a slight abuse of notation, we will use $\mathbb{B}$ to denote this 3D grid of voxels. For a voxel $v \in \mathbb{B}$, let $\varphi(v,S)$ denote the point of S that is nearest to $\hat{v}$. We discretize $\mathrm{Vor}(S) \cap \mathbb{B}$ by assuming that the entire voxel v lies in the Voronoi cell of $\varphi(v,S)$. For a point $p_i \in S$, we define its *pixelized Voronoi cell* as

$$\mathrm{PVor}_S(p_i) = \{v \mid \varphi(v,s) = p_i\}.$$

The quantity $\mu|\mathrm{PVor}_S(p_i)|$ approximates the volume of $\mathrm{Vor}_S(p_i)$. This approximation improves as we increase the resolution, i.e., $\lim_{N_s,N_t\to\infty} \mu|\mathrm{PVor}_S(p_i)| = \mathrm{Vol}(\mathrm{Vor}_S(p_i) \cap \mathbb{B})$. $\mathrm{PVor}(S)$ is the subdivision of $\mathbb{B}$ induced by $\mathrm{PVor}_S(p_i)$ for $1 \le i \le n$; see Figure 3(a).

As in [5], we want to limit the *region of influence* $R(p_i)$ of each point in S, i.e., the distance of p_i becomes infinity for points outside its region of influence. We assume $R(p_i)$ to be a "cylinder" in xyt-space of radius r and height $2r$. Namely, a point $z \in R(p_i)$ if $d(z^*, p_i^*) \le r$ and $|t_z - t_i| \le r$.[1]

[1] One could have defined the region of influence to be the sphere of radius r centered at p_i, but the cylindrical region is more meaningful in our application.

We now define the *truncated pixelized Voronoi cell* of p_i as

$$\mathrm{Vor}^{\square}_S(p_i) = \{v \mid \varphi(v,S) = p_i \wedge \hat{v} \in R(p_i)\}.$$

$\mathrm{Vor}^{\square}(S)$ is the subdivision of $\mathbb{B}$ induced by $\mathrm{Vor}^{\square}_S(p_i)$, $i \le i \le n$. Note that a voxel not lying in $R(p_i)$ for any $p_i \in S$ does not belong to the truncated Voronoi cell of any point of S.

Computing pixelized Voronoi diagram. We now describe a GPU based algorithm for computing $\mathrm{Vor}^{\square}(\mathrm{S})$, which is a generalization of the algorithm described in [5] and uses the latter as a subroutine. For $1 \le i \le n$, let $f_i : \mathbb{R}^3 \to \mathbb{R}$ be the function that measures the distance between p_i and a point z in $\mathbb{R}^3$, i.e., $f_i(z) = d(z, p_i)$. The *lower envelope* f of $\{f_1, \ldots, f_n\}$ is defined as

$$f(z) = \min_{1 \le i \le n} f_i(z),$$

which is the distance from z to its nearest neighbor in S. $\mathrm{Vor}(S)$ is the projection of the graph of f onto the xyt-space. Following the argument in [5, 11], the problem of computing $\mathrm{PVor}(S)$, $\mathrm{Vor}^{\square}(S)$ can be formulated as rendering a 4D scene on a 3D hyperplane. However GPUs are able to render only 3D scenes on a 2D plane, and they have only 2D buffers. We therefore render each 2D slice of $\mathbb{B}$ (parallel to the xy-plane) separately, as described below.

Fix a time slice τ of $\mathbb{B}$, i.e., the set of voxels with time index τ. Let Π_τ be the horizontal plane passing through the center points of the voxels of this slice. $\mathbb{B}_\tau = \mathbb{B} \cap \Pi_\tau$ is a square with an $N_s \times N_s$ 2D uniform grid induced on it. Each pixel v_τ of this 2D grid, the intersection of a voxel v with Π_τ, is a square; $\hat{v}$ is the center point of both v and v_τ. For $p_i \in S$, we can thus define its *Voronoi cell slice* on Π_τ as

$$\mathrm{PVor}_S(p_i, \tau) = \{v_\tau \in \mathbb{B}_\tau \mid v \in \mathrm{PVor}_S(p_i)\},$$

$\mathrm{PVor}_S(\tau)$ is the subdivision of $\mathbb{B}_\tau$ induced by these cell slices, which can be viewed as a discretization of the 2D slice of $\mathrm{Vor}(S) \cap \Pi_t$ inside $\mathbb{B}$; see Figure 3(a,b). Similarly we can define $\mathrm{Vor}^{\square}_S(p_i, \tau)$ and $\mathrm{Vor}^{\square}_S(\tau)$.

We now define a series of functions $f_i^\tau : \mathbb{R}^2 \to \mathbb{R}$, for $1 \le i \le n$, as

$$f_i^\tau(x,y) = f_i(x,y,\tau) = d((x,y,\tau), p_i).$$

Let $f^\tau(x,y) = \min_i f_i^\tau(x,y)$ be their lower envelope. A pixel $v_\tau \in \mathrm{PVor}_S(p_i, \tau)$ if and only if the function $f_{i,\hat{v}}^\tau$ appears on the lower envelope at the center point of v_τ, i.e., $f_i^\tau(v_\tau) = f^\tau(v_\tau)$. The graph of f_i^τ is a 2-dimensional surface γ_i. For example, if $d(\cdot,\cdot)$ is the Euclidean distance then

$$f_i^\tau(x,y) = \left((x - x_i)^2 + (y - y_i)^2 + (\tau - t_i)^2\right)^{1/2}$$

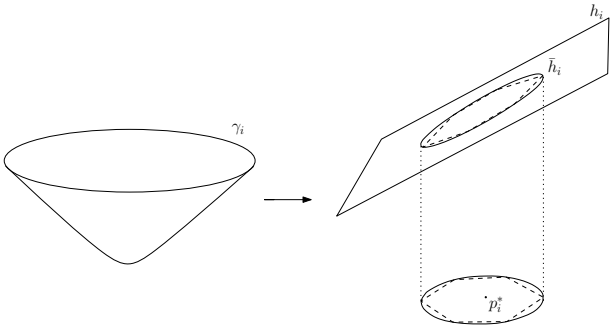


Figure 5. In the case of the Euclidian metric, γ_i can be simplified using the lifting transform.

and its graph is one sheet of a hyperboloid of revolution of two sheets, see Figure 4(a,b). As in [5], $\text{PVor}_S(\tau)$ can be computed by viewing $\mathbb{B}_\tau$ as the 2D image plane and rendering the surfaces $\gamma_1, \ldots, \gamma_n$ with $(0, 0, -\infty)$ as the viewpoint. We set the color of γ_i to i, then the color buffer $\mathbb{C}[\pi]$ stores the index of the point $\varphi(\pi, S)$, i.e., the point whose Voronoi cell contains π; see Figure 3(c). In order to compute $\text{Vor}^\square{}_S(\tau)$ we need to truncate γ_i within $R(p_i)$, the region of influence of p_i, before rendering it. By repeating this for all slices $0 \le t < N_t$, we can compute $\text{Vor}^\square(S)$. For a fixed τ, let $S^\tau = \{p \in S \mid \tau - r \le t_p \le \tau + r\}$. Then $\text{Vor}^\square(S, \tau) = \text{Vor}^\square(S^\tau, \tau)$. So while computing $\text{Vor}^\square(S, \tau)$, we only render γ_i's for $p_i \in S^\tau$.

Triangulating γ_i. As mentioned in Section 2, the GPU can render only linear objects. Therefore we need to approximate each γ_i (or its truncated version) by a triangulated surface, $\gamma_i^\diamond$. Here we describe the triangulation procedure for the case when $d(\cdot, \cdot)$ is a convex function. For technical reasons, we also assume that $d(\cdot, \cdot)$ does not have a cross term between t and x, y, i.e., we can separate spatial and temporal terms. This implies that $d(\cdot, \cdot)$ can be restricted to define the distance between two points in $\mathbb{R}^2$. We assume that γ_i is such that $|t_i - \tau| \le r$. Next, we clip γ_i within points $\pi \in \mathbb{R}^2$ s.t. $d(\pi, p_i^*) \le r$. By our assumption that there are no cross terms, all points on the boundary of the clipped surface γ_i have the same z-value. Let γ_i also denote the resulting surface patch. Next, we choose a parameter m and compute m level-set curves (contour lines) at regular height intervals on γ_i, i.e., intersect γ_i with m horizontal planes (e.g. dashed circles in Figure 4(c). Each curve is a convex closed curve, which we approximate by a convex k-gon, for some parameter k, by choosing k points on the curve. We then create triangles between adjacent k-gons using $2k$ triangles. The resulting triangulated surface $\gamma_i^\diamond$, composed of $2km$ triangles, approximates γ_i. See Figure 4(c). the approximation error can be controlled by choosing the values of m and k carefully.

The case of Euclidean distance. Finally, we show that if $d(\cdot, \cdot)$ is a weighted Euclidean distance function as defined in (3), there is a much better way of rendering γ_i using the so-called lifting transform [6], which uses fewer triangles and induces no error in the distance function (though the region of influence is still approximated)

For $1 \le i \le n$, define $g_i^\tau : \mathbb{R}^2 \to \mathbb{R}$ as

$$\begin{aligned} g_i^\tau(x, y) &= f_i^\tau(x, y)^2 - x^2 - y^2 \\ &= -2xx_i - 2yy_i + x_i^2 + y_i^2 + \omega^2(t_i - \tau)^2. \end{aligned}$$

Let

$$g^\tau(x, y) = \min_i g_i^\tau(x, y).$$

The crucial observation is that

$$g_i^\tau(x, y) \le g_j^\tau(x, y) \Leftrightarrow f_i^\tau(x, y) \le f_j^\tau(x, y),$$

which implies that the xy-projection of the lower envelope of f and g are identical, therefore $\text{PVor}_s(\tau)$ is also the projection of the graph of $g^\tau(x, y)$. We therefore render the graphs of g_i^τ instead of f_i^2. The graph h_i of g_i^τ is a 2-dimensional plane, so it can be rendered directly. In order to compute $\text{Vor}^\square(S)$, we need to truncate h_i within the region of influence of h_i. We first assume that $|t_i - \tau| \le r$. Next, let $\bar{h}_i$ be the portion of h_i clipped within the region of influence of p_i,

$$\bar{h}_i = \{(x, y, z) \in h_i \mid (x - x_i)^2 + (y - y_i)^2 \le r^2\},$$

which is an ellipse in 3D. We wish to render $\bar{h}_1, \ldots, \bar{h}_n$. We approximate $\bar{h}_i$ to a convex k-gon $h_i^\diamond$ as follows. Let σ be a regular k-gon in $\mathbb{R}^2$. We set $\sigma_i = \sigma + p_i^*$. $h_i^\diamond$ is obtained by lifting σ_i to h_i, i.e.,

$$h_i^\diamond = \{(x, y, g_i^\tau(x, y)) \mid (x, y) \in \sigma_i\}.$$

For each vertex v of σ_i there is a vertex (v, g_i^τ) in $h_i^\diamond$. We triangulate $h_i^\diamond$ into k triangles in a standard manner. Note that $h_i^\diamond$ encodes the function g_i^τ exactly; it only approximates the region of influence.

The GVor algorithm. Let $\text{GVor}(S, \tau)$ denote the algorithm, described in this section, to compute the slice of the truncated pixelated Voronoi diagram at time τ, i.e., $\text{Vor}^\square(S, \tau)$ which is the same as $\text{Vor}^\square(S^\tau, \tau)$. The output of this algorithm is the combination of color buffer $\mathbb{C}$ and depth buffer $\mathbb{D}$, which together describe $\text{Vor}^\square(S, \tau)$. However, we apply the following twist: we set the color of each triangle of $\gamma_i^\diamond$ to $h(p_i)$ (instead of i). Thus $\mathbb{C}[\pi]$ stores $h(p_i)$ for all pixels $\pi \in \text{Vor}^\square{}_S(p_i, \tau)$. We define the *frame buffer* $\mathbb{F}$ to be $(\mathbb{C}, \mathbb{D})$ and we can think of $\mathbb{F}$ as the result of running GVor, i.e., $\mathbb{F} = \text{GVor}(S, \tau)$. Ignoring initialization costs, the complexity of $\text{GVor}(S, \tau)$ is that of rendering $\gamma_i^\diamond$ (or $h_i^\diamond$) for each p_i within the region of influence.

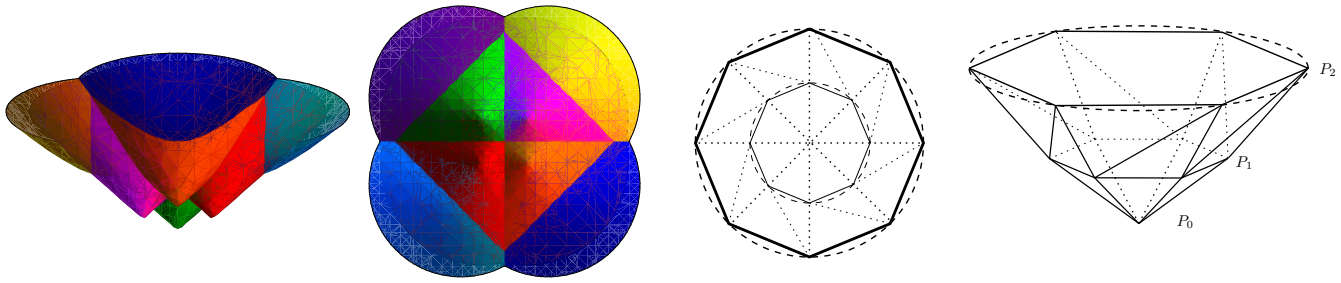
4 Natural Neighbor Interpolation on Grids

In this section we describe our algorithms for answering natural-neighbor interpolation queries on a $M \times M \times T$ grid $\mathbb{Q}$ of points in $\mathbb{R}^3$ using truncated pixelized Voronoi diagrams. We assume that $\mathbb{Q}$ has origin $(0, 0, 0)$ and resolution 1 in each dimension, i.e., $\mathbb{Q} = \{0, \ldots, M - 1\}^2 \times \{0, \ldots, T - 1\}$. Any 3D grid can be mapped to $\mathbb{Q}$ using an affine transformation and using the weighted Euclidian metric to preserve the structure of the Voronoi diagram.

Since the region of influence has radius r and height $2r$, we define $\mathbb{B} = \mathbb{Q} + [-2r - 1/2, 2r + 1/2]^3$ — large enough to contain all the points of S that are of interest, and we can assume that $S \subset \mathbb{B}$. For simplicity we assume that $r \in \mathbb{N}$. Let $s \in \mathbb{N}$ be a parameter. We discretize $\mathbb{B}$ into a $sM \times sM \times T$ 3D grid of voxels, each voxel is a box of size $\frac{1}{s} \times \frac{1}{s}$ and height 1, and its volume is $\mu = \frac{1}{s^2}$. By our definition of $\mathbb{B}$ and the voxels, each grid point of $\mathbb{Q}$ lies at the center of an $s \times s \times 1$ array of voxels of $\mathbb{B}$, i.e., the temporal resolution of $\mathbb{B}$ and $\mathbb{Q}$ is the same, but the spatial resolution of $\mathbb{B}$ is s times that of $\mathbb{Q}$. One can choose the temporal resolution of $\mathbb{B}$ higher than that of $\mathbb{Q}$ but it is not important in our applications and choosing the two to be the same simplifies the description of the algorithm.

Recall the definition of a the natural neighbor interpolation $h : S \to \mathbb{R}^3$ given in Equations (1) and (2). In our discretized setting we modify h as follows:

$$h(q) = \frac{\sum_{p \in S} |\text{Vor}^\square{}_S(p) \cap \text{Vor}^\square{}_{S \cup \{q\}}(q)| h(p)}{|\text{Vor}^\square{}_{S \cup \{q\}}(q)|} = \frac{N(q)}{D(q)}. \quad (4)$$



(a) (b) (c)

Figure 4. (a,b) Lower envelope of 8 points that were used for $\mathrm{PVor}(S)$ in Figure 3(b,c). (a) shows the hyperboloids from a side view; (b) the outer cells of the Voronoi diagram are infinite, but in this figure their sizes are limited because the hyperboloids are of a limited height. (c) The triangulation of γ_i into γ_i° using two triangle strips ($m = 2$).

Algorithm # of passes	# Copies of $\mathbb{F}$	Triangles rendered per $p \in S$ General	Weighted Euclidean
r^2	1	$O(r^2kmn)$	$O(r^2n)$
r	$2r+1$	$O(rkmn)$	$O(rn)$
1	$2(2r+1)$	$O(kmn)$	$O(n)$

Table 1. Performance of the three algorithms. $|S| = n$, each copy of $\mathbb{F}$ requires storing a color buffer and associated depth buffer on the GPU.

$D(q)$ is simply the number of voxels in $\mathrm{Vor}^\square{}_{S\cup\{q\}}(q)$. $N(q)$ is the sum of the heights represented by each voxel in $\mathrm{Vor}^\square(S)$ from the Voronoi cell of q.

We observe that the region of influence along the time axis implies that only voxels v for which $|t_v - t_q| \le r$ are relevant for $N(q)$ and $D(q)$. Thus, we only need to consider $2r+1$ slices of $\mathbb{B}$ corresponding to $\Pi_{t_q-r}, \ldots, \Pi_{t_q+r}$. Applying this to (4) we get

$$D(q) = \sum_{\tau=t_q-r}^{t_q+r} |\,\mathrm{Vor}^\square{}_{S\cup\{q\}}(q,\tau)| \qquad = \sum_{\tau=t_q-r}^{t_q+r} D(q,\tau) \qquad (5)$$

$$N(q) = \sum_{\tau=t_q-r}^{t_q+r} \sum_{\pi\in\mathrm{Vor}^\square{}_{S\cup\{q\}}(q,\tau)} h(\varphi(\pi,S)) \qquad = \sum_{\tau=t_q-r}^{t_q+r} N(q,\tau), \qquad (6)$$

where

$$N(q,\tau) = \sum_{\pi\in\mathrm{Vor}^\square{}_{S\cup\{q\}}(q,\tau)} h(\varphi(\pi,S)),$$

$$D(q,\tau) = |\,\mathrm{Vor}^\square{}_{S\cup\{q\}}(q,\tau)|.$$

We refer to Table 1 for an overview of the algorithms presented in this section.

Let H be an $M\times M\times T$ array that stores the height at all points of $\mathbb{Q}$, computed using the discretized natural-neighbor interpolation defined in (4). For a fixed time $0 \le i < T$, let $\mathbb{Q}_i$ denote the query points in time slice i and let H_i denote the corresponding time slice of H. For a fixed i, by (5) and (6), the computation of H_i involves computing $\mathrm{Vor}^\square(S^{i-r}, i-r), \ldots, \mathrm{Vor}^\square(S^{i+r}, i+r)$. Since we compute each H_τ separately, a point of S has to be processed by the GPU several times. We describe three algorithms that provide tradeoffs between the number of times each point of S is processed (which we refer to as the number of *passes*) and the space (on the GPU) used by the algorithm.

r^2-pass algorithm.. We process each $\mathbb{Q}_i$, for $0 \le i < T$, separately. Equations (5) and (6) suggest that we compute $N(q,\tau)$ and $D(q,\tau)$ for $\tau = i-r, \ldots, i+r$ and for all $q \in \mathbb{Q}_i$. For each τ we first compute $\mathbb{F}_\tau = \mathrm{GVor}(S,\tau)$, giving us a representation of $\mathrm{Vor}^\square(S,\tau) = \mathrm{Vor}^\square(S^\tau,\tau)$. For $q \in \mathbb{Q}_i$, let γ_q denote the graph of the function $d((x,y,\tau),q)$, which is a 2D surface, and let γ_q° be the linear approximation of γ_i, truncated within $R(q)$. Beutel et al [5] have described an algorithm that uses $\mathbb{F}_\tau$ and renders the surfaces $\{\gamma_q^\circ \mid q \in \mathbb{Q}_i\}$ so that the color buffer $\mathbb{C}_{i,\tau}$ encodes $\mathrm{Vor}^\square(S\cup\{q\},\tau)$ for all $q \in \mathbb{Q}_i$. Then using $\mathbb{F}_\tau$ and $\mathbb{C}_{i,\tau}$ it computes $N(q,\tau)$ and $D(q,\tau)$. We call their algoirthm $\textsc{Interpolate}(\mathbb{F}_\tau, i, \tau)$. Depending on the size of M, r, s and GPU constraints, this may require several rendering passes and an I/O-efficient partitioning algorithm; we refer to [5] for details. We assume that the $\textsc{Interpolate}$ procedure returns two $M\times M$ arrays N^∇ and D^∇ s.t. $N^\nabla[q] = N(q,\tau)$ and $D^\nabla[q] = D(q,\tau)$. The pseudo-code of the algorithm is described in Algorithm 1.

Each point $p \in S$ is passed to the computation of $O(r)$ slices of $\mathrm{Vor}^\square(S^\tau,\tau)$ for $\tau \in \{\lfloor t_p\rfloor - r, \ldots, \lceil t_p\rceil + r\}$. Furthermore each Voronoi diagram slice $\mathrm{Vor}^\square(S,\tau)$ is computed $2r+1$ times for $\mathbb{Q}_{\tau-r}, \ldots, \mathbb{Q}_{\tau+r}$. Hence each point of S is passed $O(r^2)$ times.

Algorithm 1 r^2-pass

```
for i ← 0 to T − 1 do
    H_i ← N_i ← D_i ← 0
    for τ ∈ {i − r, ... i + r} do
        𝔽 ← GVor(S^τ, τ)
        (N^∇, D^∇) = Interpolate(𝔽, i, τ)
        N_i ← N_i + N^∇, D_i ← D_i + D^∇
    H_i ← N_i/D_i
return H
```

***r*-pass algorithm**. The overall structure of the algorithm is the same as before but we save some computation by storing what we have already computed and re-using it. As before, let frame buffer $\mathbb{F}_\tau = \mathrm{GVor}(S,\tau)$ refer to the combination of $\mathbb{C}_\tau$ and $\mathbb{D}_\tau$ describing $\mathrm{Vor}^\square(S,\tau)$. To answer NNI queries for points in $\mathbb{Q}_i$ we must generate $\mathbb{F}_{i-r}, \ldots \mathbb{F}_{i+r}$ and perform the interpolation on each time-slice. When we proceed to $\mathbb{Q}_{i+1}$, we must now generate $\mathbb{F}_{i+1-r}, \ldots \mathbb{F}_{i+1+r}$. We have all of the frame buffers except the last, $\mathbb{F}_{i+1+r}$, from the previous step and thus they can be reused by $\textsc{Interpolate}$ step. We only need to call $\mathrm{GVor}(S^{i+1+r}, i+1+r)$. See Algorithm 2 for the pseudo-code. This algorithm requires storing $2r+1$ frame buffers: $2r$ to save values from the previous query grid and one to generate the new Voronoi diagram. Doing this for $\mathbb{Q}_i$ for all $0 \le i < M$ *in order* lets us generate each $\mathbb{F}_i$ only once. This method renders each point p $O(r)$ times: once for each $\mathrm{Vor}^\square(S^\tau,\tau)$ for $\tau \in \{\lfloor t_p\rfloor - r, \ldots, \lceil t_p\rceil + r\}$.

1-pass algorithm. We now show that the number of passes can be reduced to 1 if S is time series data and $d(\cdot,\cdot)$ satisfies certain

Algorithm 2 r-pass

Create $\mathbb{F}_{-r-1} \dots \mathbb{F}_{r-1}$
for $i \leftarrow 0$ to $T-1$ **do**
  $H_i \leftarrow N_i \leftarrow D_i \leftarrow 0$
  Delete $\mathbb{F}_{i-r-1}$ from GPU
  $\mathbb{F}_{i+r} \leftarrow \text{GVor}(S^{i+r}, i+r)$
  for $\tau \in \{i-r, \dots i+r\}$ **do**
    $(N^\nabla, D^\nabla) = \text{Interpolate}(\mathbb{F}_\tau, i, \tau)$
    $N_i \leftarrow N_i + N^\nabla$, $D_i \leftarrow D_i + D^\nabla$
  $H_i \leftarrow N_i / D_i$
return H

properties and the GPU supports certain primitives. For simplicity, we assume that $S = \Sigma^1 \cup \Sigma^2 \cup \cdots$, where the t-coordinate of all points in Σ^i is i, the time coordinate of the ith slice of $\mathbb{Q}$, and $d(\cdot,\cdot)$ is the weighted Euclidian distance. The first assumption can easily be removed. The second assumption can be relaxed to a larger family of distance functions, but identifying this family is technical and omitted from this version. Recall that $\text{Vor}(\Sigma^i, j)$ is the projection of the lower envelope $g^i_j(x,y)$ of the functions

$$g^i_{j,p}(x,y) = -2xx_p - 2yy_p + x_p^2 + y_p^2 + \omega^2(j-i)^2 \quad \forall p \in \Sigma^i \qquad (7)$$

Since the last term in (7) does not depend on p, it immediately implies that $\text{Vor}(\Sigma^i, j)$ is the same for all j and that

$$g^i_j(x,y) = g^i_i(x,y) + \omega^2(j-i)^2.$$

Let $\mathbb{F}^i_j = (\mathbb{C}^i_j, \mathbb{D}^i_j) = \text{GVor}(\Sigma^i, j)$ — $\mathbb{C}^i_j[\pi]$ stores the height of the point nearest to π and $\mathbb{D}^i_j$ stores the distance from π to its nearest neighbor, and let $\mathcal{F}^i = \mathbb{F}^i_i$. Then $\mathbb{F}^i_j$ can be computed from $\mathcal{F}^i$ by adding the value $\omega^2(j-i)^2$ to every pixel in $\mathbb{D}^i$. Let $\text{Shift}(\mathbb{F}, \delta)$ be the procedure that adds the value δ to all pixels in $\mathbb{D}$. Then

$$\mathbb{F}^i_j = \text{Shift}(\mathcal{F}^i, \omega^2(j-i)^2).$$

Recall that $S^j = \bigcup_{\ell=-r}^{r} \Sigma^{j+\ell}$. Assuming we have $\mathcal{F}^\ell$ for $j-r \le \ell \le j+r$, we can compute the frame buffer $\mathbb{F}_j = (\mathbb{C}_j, \mathbb{D}_j)$ corresponding to $\text{Vor}^\square(S^j, j)$, using the procedure $\text{Combine}(\mathcal{F}^{j-r}, \dots, \mathcal{F}^{j+r})$ as follows. We first compute $\mathbb{F}^\ell_j$ for $j-r \le \ell \le j+r$, and set

$$\mathbb{D}_j[\pi] = \min_{j-r \le \ell \le j+r} \mathbb{D}^\ell_j[\pi]$$

and finally set $\mathbb{C}_j[\pi]$ to the contents of $\mathbb{C}^\ell_j[\pi]$ if $\mathbb{D}_j[\pi] = \mathbb{D}^\ell_j[\pi]$. The pseudo-code can be found in Algorithm 3.

Algorithm 3 $\text{Combine}(\mathcal{F}^{j-r}, \dots, \mathcal{F}^{j+r})$

for all $\pi \in \mathbb{F}_j$ **do**
  $\mathbb{C}_j[\pi] \leftarrow 0$, $\mathbb{D}_j[\pi] \leftarrow \infty$
for $i \in j-r, \dots, j+r$ **do**
  $\mathbb{F}^i_j \leftarrow \text{Shift}(\mathcal{F}^i, \omega^2(j-i)^2)$
  for all $\pi \in \mathbb{F}_j$ **do**
    if $\mathbb{D}^i_j[\pi] < \mathbb{D}_i[\pi]$ **then**
      $\mathbb{D}_j[\pi] \leftarrow \mathbb{D}^i_j[\pi]$
      $\mathbb{C}_j[\pi] \leftarrow \mathbb{C}^i_j[\pi]$
return $\mathbb{F}_j$

With these procedures at hand, we modify the r-pass algorithm as follows, so that each point is processed once. See Algorithm 4 for the pseudo-code. The algorithm maintains the invariant that while processing $\mathbb{Q}_i$, it maintains $\mathbb{F}_{i-r}, \dots, \mathbb{F}_{i+r}$ and also $\mathcal{F}^i, \dots, \mathcal{F}^{i+2r}$. While processing of $\mathbb{Q}_i$, the algorithm first computes $\mathbb{F}^{i+2r}$ using $\text{GVor}(\Sigma^{i+2r}, i+2r)$. Next, instead of computing $\mathbb{F}_{i+r}$ by invoking

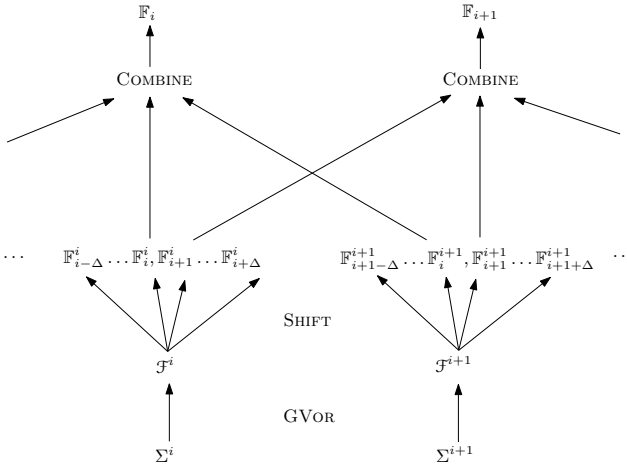


Figure 6. The process of creating $\mathbb{F}_i$ from sets of data Σ^i rendering each point only once.

$\text{GVor}(S^{i+r}, i+r)$, it uses the procedure $\text{Combine}(\mathcal{F}^i, \dots, \mathcal{F}^{i+2r})$. The rest of the procedure is the same. Note that each point is processed only once.

Algorithm 4 1-pass

Create $\mathcal{F}^{-1} \dots \mathcal{F}^{2r-1}$ and $\mathbb{F}_{-r-1} \dots \mathbb{F}_{r-1}$
for $i \leftarrow 0$ to $T-1$ **do**
  $H(\mathbb{Q}_i) \leftarrow N(\mathbb{Q}_i) \leftarrow D(\mathbb{Q}_i) \leftarrow 0$
  Delete $\mathcal{F}^{i-1}$ and $\mathbb{F}_{i-r-1}$ from GPU
  $\mathcal{F}^{i+2r} \leftarrow \text{GVor}(\Sigma^{i+2r}, i+2r)$
  $\mathbb{F}_{i+r} \leftarrow \text{Combine}(\mathcal{F}_i \dots \mathcal{F}_{i+2r})$
  for $\tau \in \{i-r, \dots i+r\}$ **do**
    $(N^\nabla, D^\nabla) = \text{Interpolate}(\mathbb{F}_\tau, i, \tau)$
    $N_i \leftarrow N_i + N^\nabla$, $D_i \leftarrow D_i + D^\nabla$
  $H_i \leftarrow N_i / D_i$
return H

5 Experimental Results

We now discuss the details of the implementation and testing of TerraNNI. It is implemented in C++ and use OpenGL for rendering on the GPU. We ran our experiments on an Intel Core2 Duo CPU E6850 at 3.00GHz with 8GB of internal memory. We used Ubuntu 10.4 and two 1TB SATA disk drives in a RAID0 configuration. Additionally, the machine contained a NVIDIA GeForce GTX 470 graphics card running CUDA 3.0. This card has 1.2GB of memory, 448 CUDA cores, and 14 multiprocessors.

Data sets. We ran TerraNNI on two data sets. We performed a majority of our tests on LiDAR data collected from the Nags Head, NC region for years 1997-1999, 2001, 2004, 2005, 2007, and 2008. The data ranges from 100,000 points per year to nearly 3 million points per year and comes to a total of just under 20 million points. This data set takes up 382 megabytes on disk. We will refer to this data set as the NC coastal data set. The data is freely available at [1] and was provided to us by Helena Mitasova.

We also performed tests on a much larger LiDAR data set collected from Fort Leonard Wood in Missouri (data courtesy of the U.S. Army Corps of Engineers) in 2009 and 2010. The data set consists of approximately 450 million data points over an 80.5km^2 region and takes up 8.8GB on disk.

Algorithm	r^2		r	
Shape Rendered	Hyperboloid	Plane	Hyperboloid	Plane
Binning Time	17.89s	17.53s	13.98s	17.2s
GVor(S)	8m 15s	3m 16s	1m 24s	35.8s
Draw Query Cones	4m 1s	2m 26s	3m 44s	2m 28s
Interpolate	3m 45s	4m 18s	2m 45s	3m 5s
Write points	5.2s	4.54s	3.47s	4.53s
Total running time	16m 26s	10m 23s	8m 11s	6m 32s

Table 2. Comparison of running times of different variants of our algorithm on the NC coastal data set.

Parameter choices. Within our experiments there are numerous parameters that can be adjusted to tune our TerraNNI's speed and quality trade-offs. Many of these parameters stem from the 2D algorithm [5] and we refer to that paper for further details. We use a weighted Euclidean metric as the distance function for all our tests. The speed of the algorithm is highly dependent on the number of triangles rendered. In rendering the weighted Euclidean metric we tested rendering hyperboloids γ_i as well as planes h_i. In rendering each hyperboloid we set $k = 50$ and $m = 5$ such that each $\gamma_i^\diamond$ was composed of 500 triangles. As discussed in Section 3, when working with the Euclidean metric we can apply the lifting transform to render planes rather than hyperboloids. When rendering planes, we clip the plane h_i to an ellipse $\bar{h}_i$, we approximate it by a rectangle $h_i^\diamond$ circumscribing $\bar{h}_i$, and partition $h_i^\diamond$ into two triangles.

Efficiency. The largest data set in our tests was the Ford Leonard Wood data set which covers two years. Here we created a 3D grid with a spatial resolution of 5m (a total of 4 million query points) and positioned in time halfway between the 2009 and 2010 surveys. This run took 26 minutes to complete.

The NC coastal data set is interesting due to the larger temporal range of the data. We used this data set to compare various trade-offs in the r-pass and r^2-pass algorithms. We ran different variants of TerraNNI, every time producing a 3D grid with at a spatial 1 meter resolution over a $2.3 \times 1.8\text{km}^2$ region. The 3D grid has 12 slices in time, corresponding to producing an output every half year for 6 years in total. This 3D grid has a total of about 48 million voxels. The results can be found in Table 2. The table shows how much time TerraNNI spends in each phase, and include timings for the variant using the lifting transform where we only have to render a plane (two triangles) per input and query point. As can be seen in the table, the r-pass algorithm takes 6 minutes and 32 seconds using planes, approximately 60% of the time of the r^2-pass algorithm, and takes 8 minutes and 11 seconds when using hyperboloids, 50% of the time of the r^2-pass algorithm. We save between 40% and 55% of the rendering time by using planes rather than hyperboloids since the planes consist of significantly less triangles.

As indicated by Table 2 the Interpolate procedure takes longer for planes than hyperboloids. As described above, the way we clip and approximate hyperboloids and planes γ_i and h_i to obtain $\gamma_i^\diamond$ and $h_i^\diamond$, the xy-projection of the latter covers a larger area in the xy-plane, and thus cover more pixels. Consequently Interpolate performs additional atomic operations in this case (see [5] for details). By approximating $\bar{h}_i$ with a polygon $h_i^\diamond$ with more vertices (k) and thus increasing the number of triangles into which $h_i^\diamond$ is decomposed, one can achieve a trade-off between the time spent by the GVor and Interpolate procedures and choose a value of k that minimizes the overall running time.

We also compared our implementation to traditional (exact) CPU algorithms. Note that, in addition to being confined to the CPU, these algorithms also attemp to robustly and exactly compute the Voronoi diagrams. We used the NC Coastal data set as a starting point for these tests but also tried with smaller synthetic data sets. Common for all the algorithms below is that they have problems storing the entire Voronoi diagram in memory, and in practice this means that the available system memory decides how large data sets they can handle. Furthormore, the NC Coastal data set is a time series, i.e., the points have one of 8 different time values. Thus there are millions of coplanar points which causes robustness issues for exact algorithms and we often had to randomly perturb the points in the time axis to work around this issue. Spatially, the points tend to cover the same area as well, leading to further robustness issues.

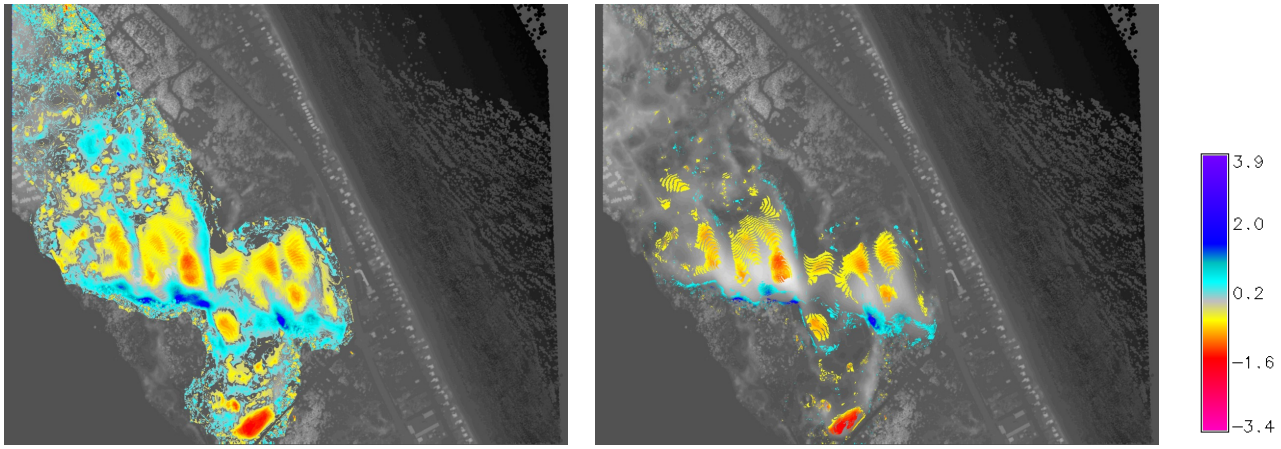
The Interpolate3d [10] package is a freely available 3D NNI implementations for performing NNI of 3D vector fields on the CPU. Unfortunately it does not run on the full NC Coastal data set, exhausting the 8GB of system memory after 2 hours, about halfway during the construction of the Voronoi diagram of the input points. It also runs out of memory interpolating 15 million points (the NC Coastal data set has 20 million points). But it is able to complete with 10 million input points, and interpolating at 50,000 query points in about three hours, with the memory usage peaking at 7.8GB. Compare this to the r-pass algorithm using planes that interpolates at 48 million query points using the full 20 million point set in less than 7 minutes. The Interpolate3d algorithm had issues with the coplanar points of the original data set and we perturbed the points in time to complete the test.

We also compared TerraNNI against state of the art CPU-based algorithms for computing the Delaunay triangulation, the dual of the Voronoi diagram. This test is relevant since this computation is a fundamental part of computing NNI. We used the algorithm from the Cgal [2] library as well as the one available in Qhull [4]. Cgal was able to successfully compute the Delaunay triangulation of the NC coastal data set, using took 11 minutes and with a peak memory use of just under 7GB. Impressively, it ran on the original unperturbed data set. However, this is significantly slower than TerraNNI which takes six and a half minutes to do the full interpolation and only 35.8 seconds of this time is spent on constructing the pixelized truncated Voronoi diagram. Additionally the Cgal algorithm is already hitting the limit on our system memory and was unable to successfully triangulate the full Fort Leonard Wood data set.

Qhull [4] offers a variety of strategies for dealing with degeneracies, we choose to perturb the input points along the time axis. Qhull runs out of memory after a couple of minutes on the full data set, the largest data set we were able to handle with a 8GB of system memory was 3 million points which took 3 and a half minutes with a peak memory use of 6GB. With 4 milion points the memory used exceeded the 8GB of system memory and the computation time increased dramatically to one and a half hour.

6 Application to Change Analysis

In this section we describe an application of TerraNNI to geomorphologic analysis of multi-temporal NC coastal LiDAR data.



(a) Regression slope values for $c_{min} = 0$ (b) Regression slope values for $c_{min} = 0.85$

Figure 7. Regression line slope values from year 2000 to 2006 superimposed on year 2000 DEM.

Figure 1(c) gives a map of Nags Head, NC. The focus of our application study is Jockey's Ridge state park sand dune. Specifically, we investigate the annual sand dune growth and erosion from years 2000 to 2006. In order to perform the study, first we apply TerraNNI to interpolate a digital elevation model (DEM) for each year from 2000 to 2006, inclusively, since many of the yearly collects (e.g., 2000, 2002, and 2003) are not available. We set the radius of influence to 2. Second, we estimate a linear regression model for each pixel and all of its temporal neighbors (i.e., years 2000 to 2006). For example, if $p(x, y, t)$ is the pixel value (height) at location (x, y) and time t, a linear regression model is estimated for the set $\{p(x, y, t_1), \ldots, p(x, y, t_n)\}$. Third, we apply the change analysis method of [16] by examining the rates of change (i.e., slopes) of the model estimates for pixel observations that exhibit strong linear trends. As in [16], we define the notion of a *coefficient of determination* χ: a measure of the reliability of the linear model in predicting the pixel values (response variable), $\chi \in [0, 1]$ is expressed as follows: $\chi = 1 - (SS_{\text{err}}/SS_{\text{tot}})$ where SS_{err} is the sum of squared errors between the linear model's estimates and response variable and SS_{tot} is the total sum of squared deviations of the response variable to its expectation [7]. A high χ implies that the SS_{err} is relatively small compared to SS_{tot}, indicating that the linear model can well predict the sample observations. We set a threshold value c_{min} and choose those pixels for which $\chi \geq c_{\text{min}}$. The threshold c_{min} can be regarded as the minimum proportion of the variation in the response variable that is captured by the linear model.

Figure 7(a) depicts the slope values for the entire sand dune region for $c_{\text{min}} = 0$. Strong erosions (negative slopes) are observed in the northwest region with extensive accumulation (positive slopes) in the southeast area of the dune. This erosion and accumulation suggest a net wind force pattern movement towards the southeast direction; however, these observed slope values do not convey additional information on the nature of the movements throughout the years. But by using the linear regression estimates, further insights into the inter-annual patterns can be harnessed by extracting those composite movements that exhibit a linear trend. We can determine these linearly dependent patterns by pruning those DEM pixels that have large deviations from the regression model, which is achieved by increasing c_{min}. Figure 7(b) shows the slope values for $c_{\text{min}} = 0.85$. In this figure, we can see regions of the dune which have undergone continuous changes following a predominantly linear trend. We exemplify these changes for year 2002 to 2004 in Figure 8. From 2002 to 2003, the measured total sand growth and erosion are 20512 m^3 and 82714 m^3, respectively. For years 2003 to 2004, similar level of dune activity is observed with total growth and erosion volumes of 33895 m^3 and 74809 m^3, respectively. Because the proposed NNI approach can efficiently interpolate the missing DEMs in both space and time, geomorphologic analyses such as the one demonstrated in this case study can be rapidly and effectively executed on large areas and at high resolutions.

7 Conclusion

In this paper we have presented three algorithms for performing (discretized) NNI on a 3D grid using a GPU. The three algorithms provide different tradeoffs between GPU computational complexity and GPU memory requirement ranging from storing one buffer on the GPU and processing each input point a quadratic (in r) number of times, to processing each input point one time, but with memory use linear in the time region of influence. Furthermore, we used the lifting transform to limit the polygonal complexity of the surfaces used for each point, shifting the bottleneck away from rendering triangles. Our algorithm, and its implementation, scale to very large data sets and the underlying discretization works around the robustness problems that add additional complexity to existing CPU-based algorithms. We are unaware of any robust implementation of NNI for points in $\mathbb{R}^3$ that can handle large data sets.

Our experimental results gives an example of an application of our algorithm and suggests that is practical in real world situations, outperforming exact CPU algorithms on medium sized data sets and enabling the efficient processing on even larger data sets with limited memory requirements. We plan on making our implementation of TerraNNI publicly available.

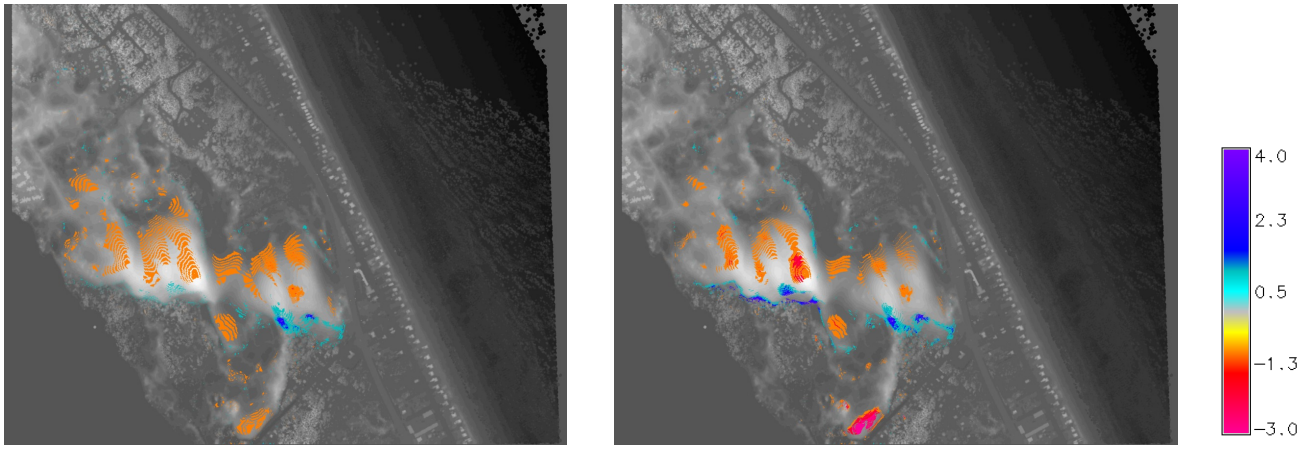
Acknowlegdements

The authors thank Helena Mitasova and the U.S. Army Corps of Engineers for access to data and and Tamal Dey and Danny Halperin for helpful discussions.

References

[1] NOAA coastal services center, LIDAR data retrieval tool. http://csc-s-maps-q.csc.noaa.gov/dataviewer/viewer.html?keyword=lidar.

[2] CGAL, Computational Geometry Algorithms Library. http://www.cgal.org.



(a) Changes from 2002 to 2003 superimposed on 2002 DEM (b) Changes from 2003 to 2004 superimposed on 2003 DEM

Figure 8. Yearly changes (meters) for areas with regression model coefficient of determination ≥ 0.85.

[3] D. Attali and J.-D. Boissonnat. A linear bound on the complexity of the Delaunay triangulation of points on polyhedral surfaces. *Discrete & Computational Geometry*, 31:369–384, 2004.

[4] C. B. Barber, D. P. Dobkin, and H. Huhdanpaa. The quickhull algorithm for convex hulls. *ACM Trans. on Mathematical Software*, 22(4):469–483, 1996.

[5] A. Beutel, T. Mølhave, and P. K. Agarwal. Natural neighbor interpolation based grid DEM construction using a GPU. In *Proc. 18th ACM SIGSPATIAL International Symposium on Advances in Geographic Information Systems*, pages 172–181, 11 2010.

[6] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry – Algorithms and Applications*. Springer Verlag, 1997.

[7] N. Draper and H. Smith. *Applied Regression Analysis*. Wiley-Interscience, 3rd edition, 1998.

[8] Q. Fan, A. Efrat, V. Koltun, S. Krishnan, and S. Venkatasubramanian. Hardware-assisted natural neighbor interpolation. In *Proc. 7th Workshop on Algorithm Engineering and Experiments (ALENEX*, 2005.

[9] S. Ghosh, A. E. Gelfand, and T. Mølhave. Attaching uncertainty to deterministic spatial interpolations. *Statistical Methodology*, 2011. In Print.

[10] R. Hemsley. Interpolation on a magnetic field. 2009. http://code.google.com/p/interpolate3d/.

[11] K. E. Hoff, III, J. Keyser, M. Lin, D. Manocha, and T. Culver. Fast computation of generalized Voronoi diagrams using graphics hardware. In *Proc. 26th Annual Conference on Computer Graphics and Interactive Techniques*, pages 277–286, 1999.

[12] P. C. Kyriakidis and A. G. Journel. Geostatistics space-time models: A review. *Mathematical Geology*, 31(6):651–684, 1999.

[13] L. Li and P. Revesz. A comparison of spatio-temporal interpolation methods. In *Geographic Information Science*, volume 2478, pages 145–160. Springer, 2002.

[14] J. Mateu, F. Montes, and M. Fuentes. Recent advances in space-time statistics with applications to environmental data: An overview. *Geophysical Research*, 108(8774), 2003.

[15] E. Miller. Towards a 4D-GIS: four dimensional interpolation utilizing kriging. In Z. Kemp, editor, *Innovations in GIS 4*, pages 181–197. 1997.

[16] H. Mitasova, E. Hardin, M. Overton, and R. Harmon. New spatial measures of terrain dynamics derived from time series of lidar data. In *Proc. 17th International Conference on Geoinformatics*, pages 1–6, 2009.

[17] H. Mitasova, L. Mitas, W. M. Brown, D. P. Gerdes, I. Kosinovsky, and T. Baker. Modeling spatially and temporally distributed phenomena: new methods and tools for GRASS GIS. *International Journal of Geographical Information Systems*, 9(4):433–446, 1995.

[18] H. Mitasova, M. Overton, and R. S. Harmon. Geospatial analysis of a coastal sand dune field evolution: Jockey's ridge, north carolina. *Geomorphology*, 72(1-4):204 – 221, 2005.

[19] T. Mølhave, P. K. Agarwal, L. Arge, and M. Revsbæk. Scalable algorithms for large high-resolution terrain data. In *Proc. 1st International Conference and Exhibition on Computing for Geospatial Research & Application*. ACM, 2010.

[20] NVIDIA. CUDA homepage. http://nvidia.com/cuda, 2010. 3.0.

[21] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[22] S. Shekhar and H. Xiong, editors. *Encyclopedia of GIS*. Springer, 2008.

[23] R. Sibson. A brief description of natural neighbour interpolation. In V. Barnet, editor, *Interpreting Multivariate Data*, pages 21–36. John Wiley & Sons, Chichester, 1981.

[24] C. K. Wikle, L. M. Berliner, and N. Cressie. Hierarchical Bayesian space-time models. *Environmental and Ecological Statistics*, 5:117–154, 1998.